State Knowledge Representation in the Mission Data System

Daniel Dvorak and Robert Rasmussen
Jet Propulsion Laboratory
California Institute of Technology

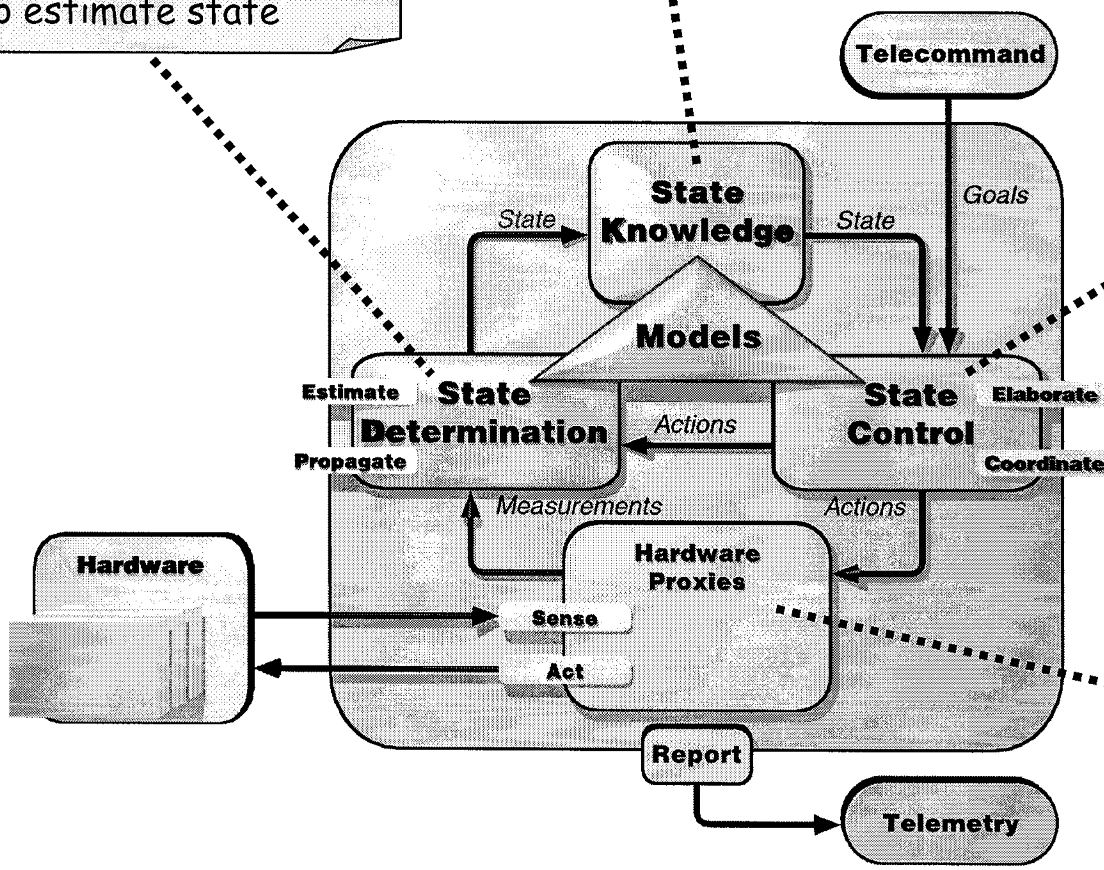
State Architecture

State variables provide access to state knowledge

Estimators interpret evidence to estimate state

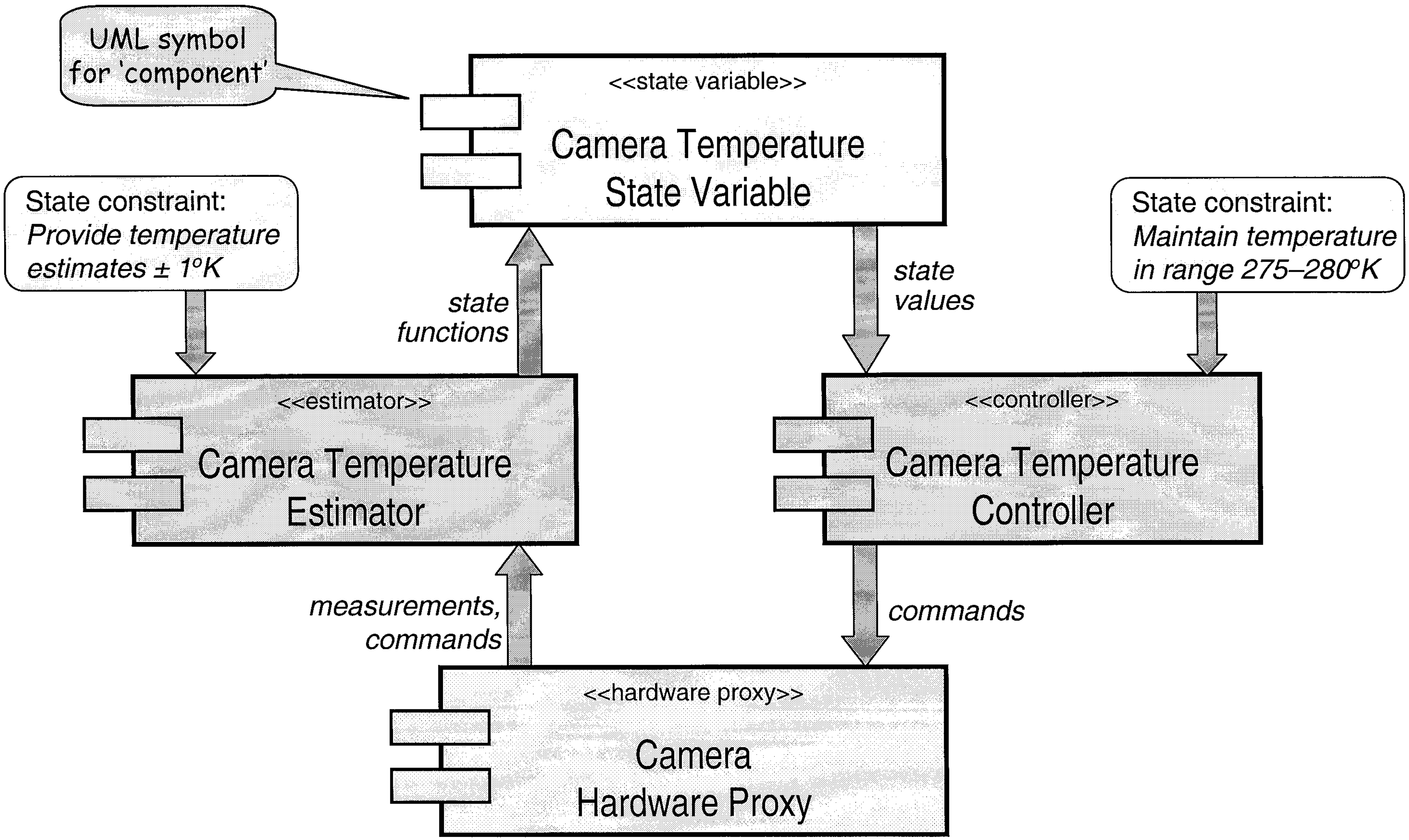
Controllers issue commands striving to achieve goals

Hardware proxies provide access to hardware busses, devices, instruments



Simple Control System Pattern

Data Flows



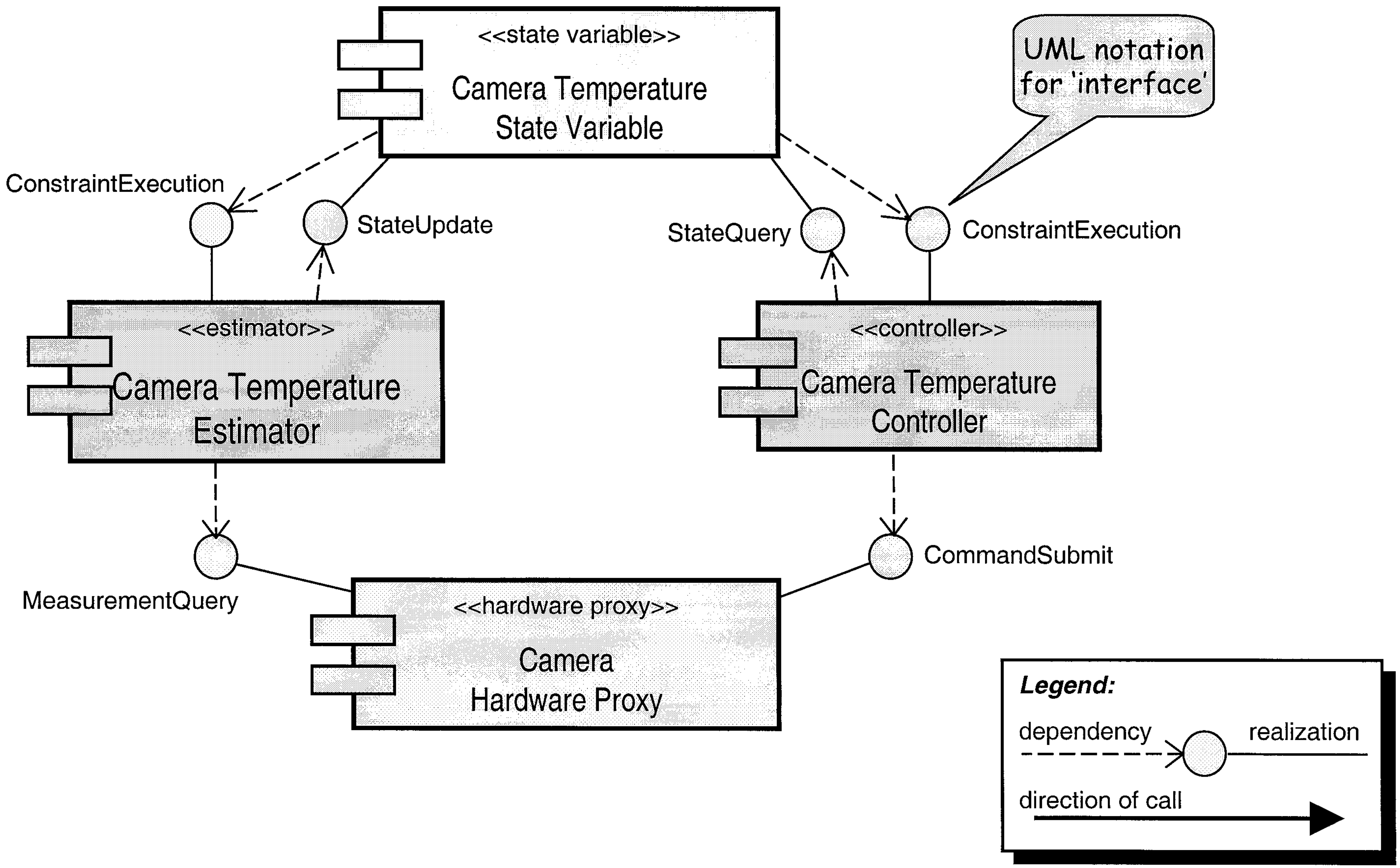
Next: Architectural Observations

Architectural Observations

- All users of state knowledge get it from state variables
  - one version of the 'truth', no private estimations
- Estimation and control are separated
  - easier to understand, less chance of error
- Estimators and controllers are both constraint-driven
  - temperature controller achieves a specified temperature range
  - temperature estimator achieves a specified quality of state knowledge
- These four components interact through shared interfaces

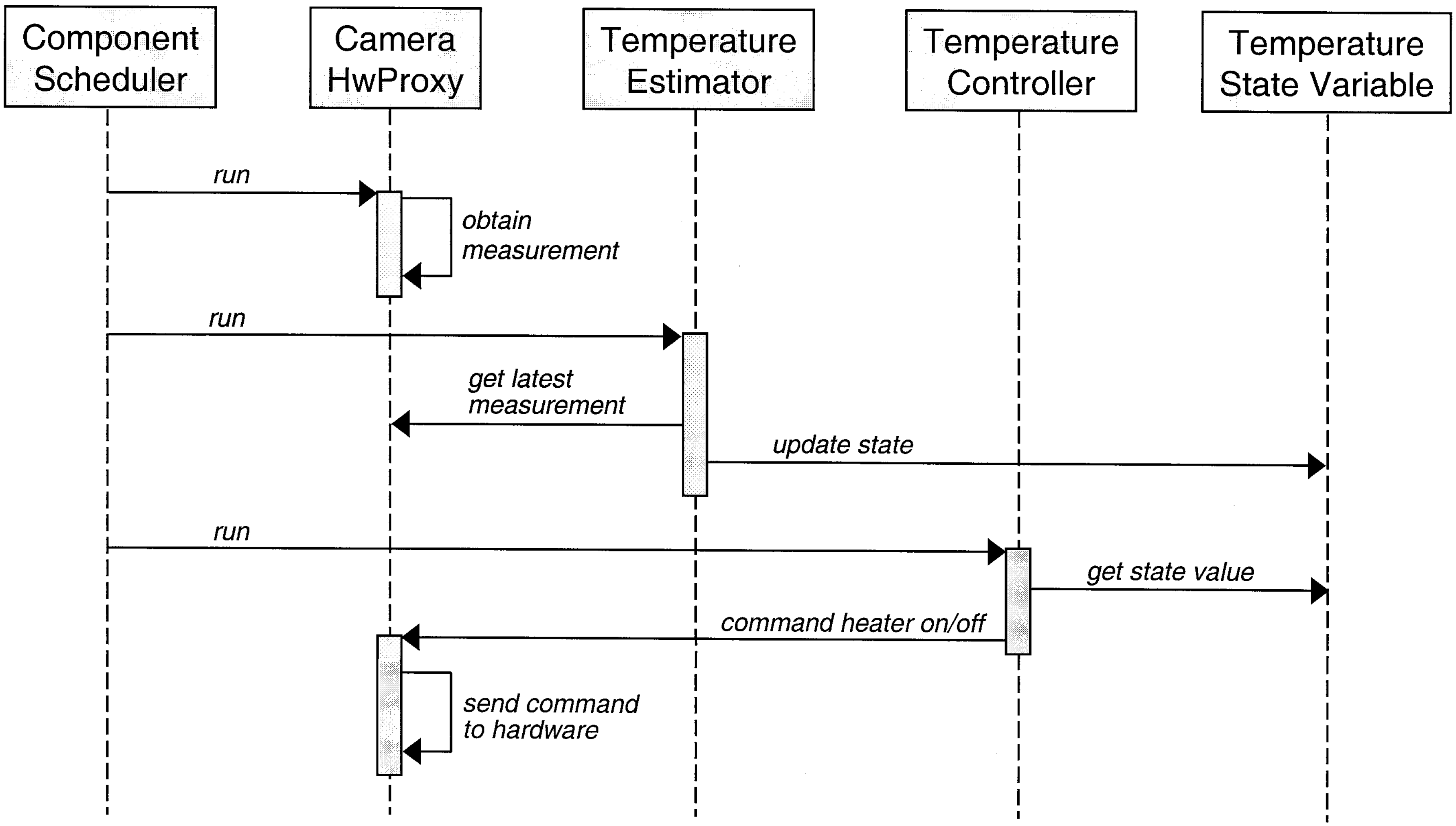
Next: Shared interfaces

Shared Interfaces



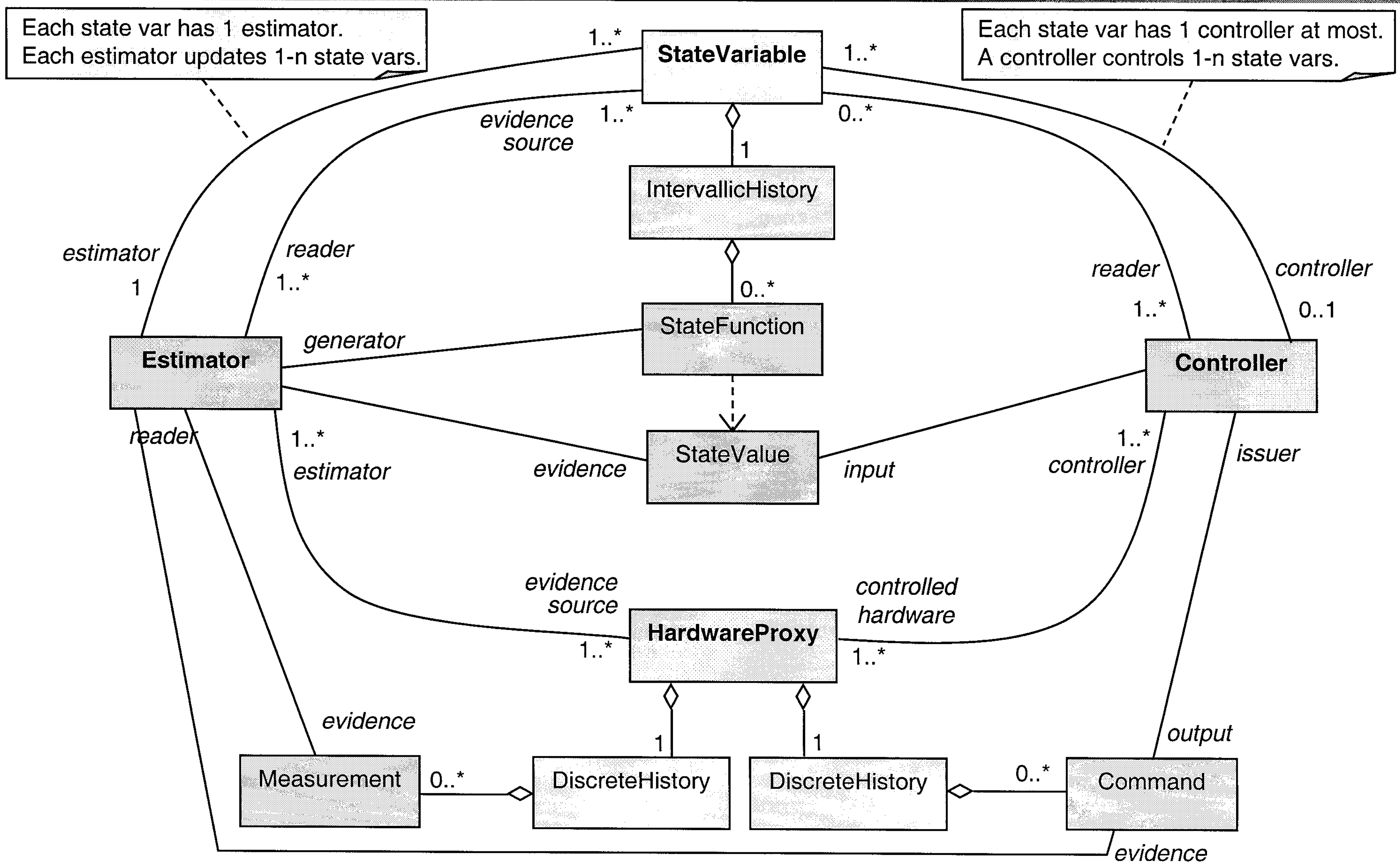
Typical Interactions

UML sequence diagram



Next: Entities & relationships

Entities & Relationships



Next: state knowledge

State Knowledge

State variables, state functions, state values

Next: scope of state knowledge

State Knowledge
"Everything you need to know"

- Dynamics
  - Vehicle position & attitude, gimbal angles, wheel rotation, …
- Environment
  - Ephemeris, light level, atmospheric profiles, terrain, …
- Device status
  - Configuration, temperature, operating modes, failure modes, …
- Parameters
  - Mass properties, scale factors, biases, alignments, noise levels, …
- Resources
  - Power & energy, propellant, data storage, bandwidth, …
- Data product collections
  - Science data, measurement sets, …
- Data Management/Data Transport Policies
  - Compression/deletion, transport priority, …
- Externally controlled factors
  - Space link schedule & configuration, …
- … and so on

Next: true state vs. state estimate

State Knowledge

- "True state" versus "state estimate"
  - a physical system has state
    - camera temperature, battery voltage, switch position, ...
  - can never know states with complete accuracy or certainty
    - only a simulator knows state values precisely
  - best we can do is *estimate* the state
  - estimates *are* state knowledge
    - it is what you know *and* how well you know it

- State variables provide access to state knowledge
  - estimates are the values of state variables
  - state variable contains a *timeline* of past, present, and future
  - to get estimate of a particular state, ask its state variable
  - "Grand Central Station" in the architecture
    - real-time estimation & control, deliberative planning & scheduling

Next: Evolution of a state variable

Evolution of State Knowledge Design

"In the beginning ..."

```
...
// camera temperature state var
double cam_temp;
...
// update temperature value
cam_temp = func1();
...
// use temperature value
func2(cam_temp);
...
```

Improvements:

- units (e.g. Kelvin)
- telemetry
- remote access
- represent uncertainty (e.g. mean & variance)
- persistent storage
- notify upon change
- access control
- startup initialization
- identify by name, for queries and for goals
- represent value as a function of time

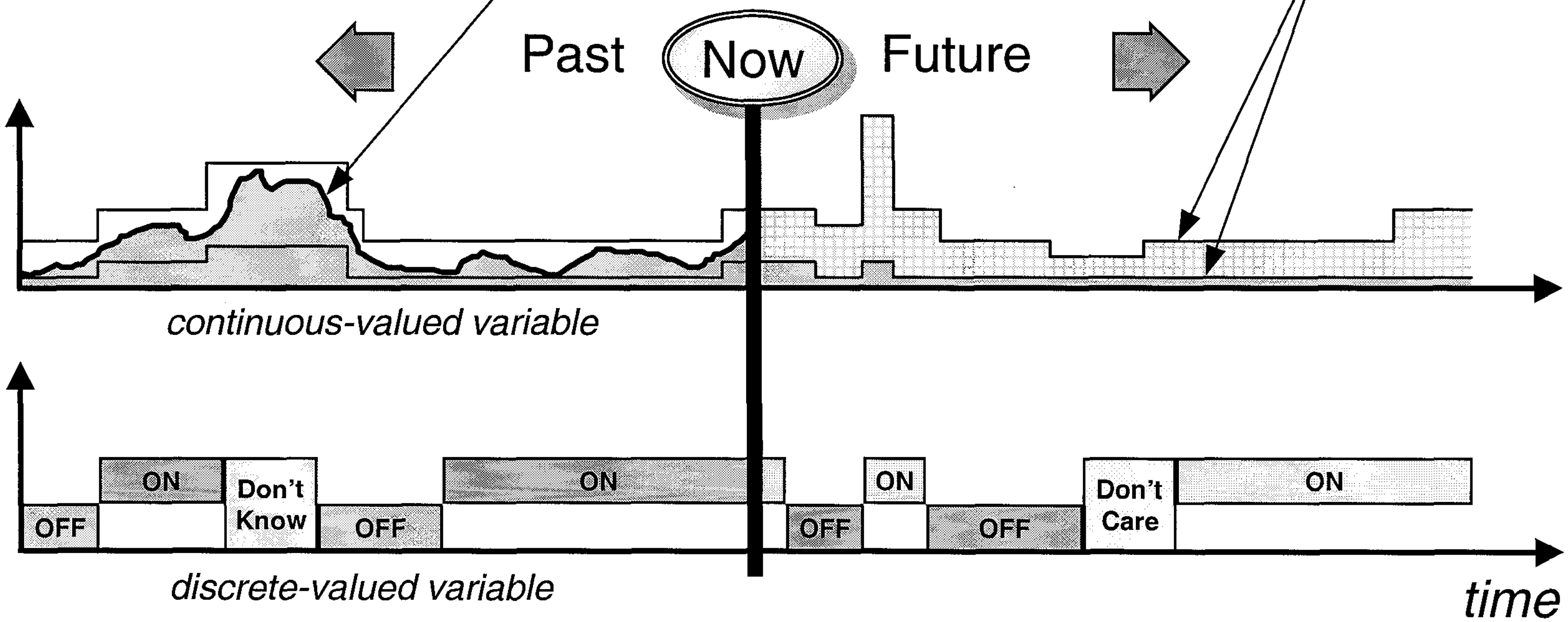
These factors shaped the state knowledge package

State Knowledge Timeline

A timeline represents a state variable's value as a function of time

Estimated value over time is represented in a series of state functions

Acceptable values over time is represented in state constraints (from goals)

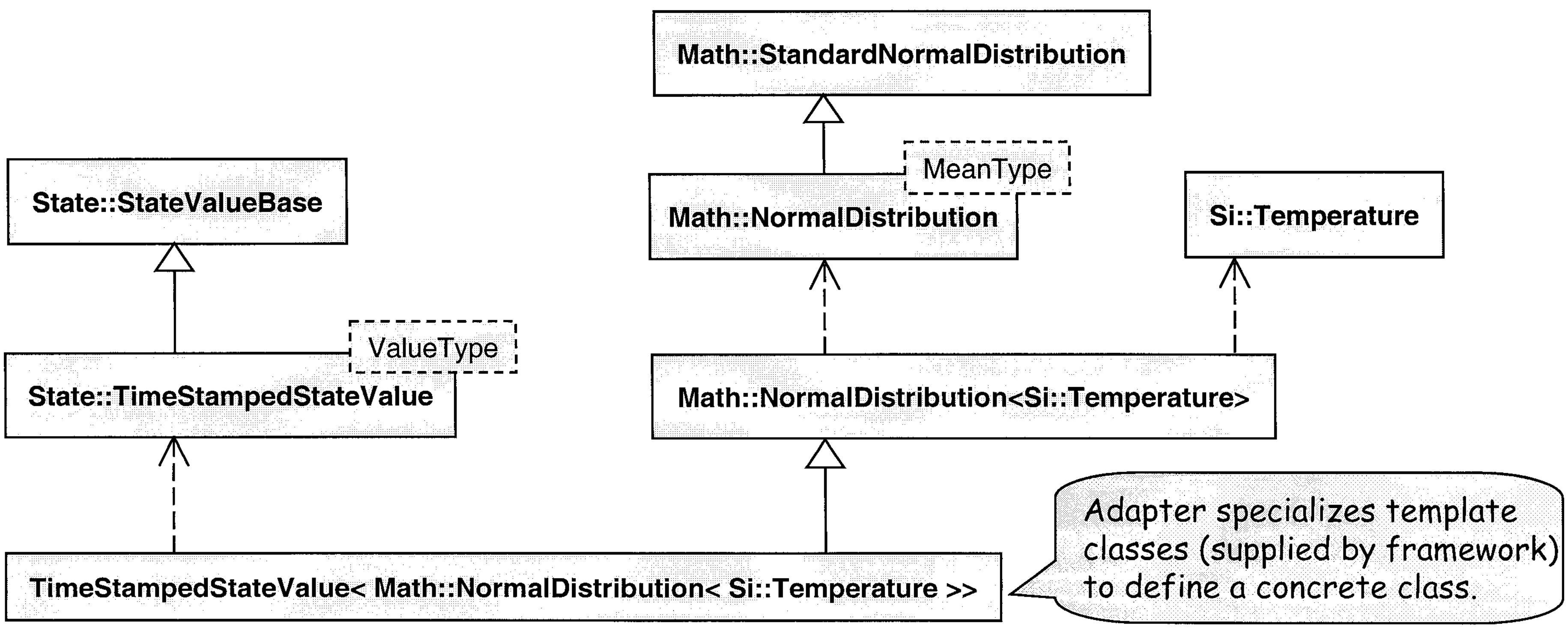


State as a function of time

- **Q**: Why represent state knowledge history as a function of time? Isn't it enough to keep a history of time-stamped samples?
- **A_1**: MDS strives to be true to the underlying physics. A physical system's state is a function of time, defined at every instant.
- **A_2**: Real-time applications become less sensitive to jitter and cycle-slip because they can obtain estimates for the current instant of time, as opposed to some pre-scheduled instant.
  - Cassini uses interpolation functions for this purpose
- **A_3**: Functions of time can be compressed in a variety of ways that conserve memory while preserving useful information.

State Value / Estimate

- *Adaptation step 1:*
  Decide how to represent a state value, including uncertainty, and decide what kinds of questions it must answer
- *Example:*
  Represent temperature in Kelvin, using mean and standard deviation, with access to timestamp, mean, and standard deviation



State Value: Inherited Functionality

Shows attributes and operations inherited from framework classes

TimeStampedStateValue< Math::NormalDistribution< Si::Temperature >>
Attributes - m_timestamp : Tmgt::RTEpoch - m_mean : Si::Temperature - m_stddev : Si::Temperature
Normal constructor, given time and temperature + TimeStampedStateValue(const RTEpoch& time, const NormalDistribution<Si::Temperature>& temp) ***Accessors for time, mean, standard deviation*** + getTime() : Tmgt::RTEpoch + getMean() : Si::Temperature + getStdDev() : Si::Temperature ***Compute probability that temperature is within a given range*** + getProbability(Si::Temperature low, Si::Temperature high) : double ***Serialization and deserialization*** + writeObject(Ser::ObjectOutputStream&) : void + TimeStampedStateValue(Ser::ObjectInputStream&)

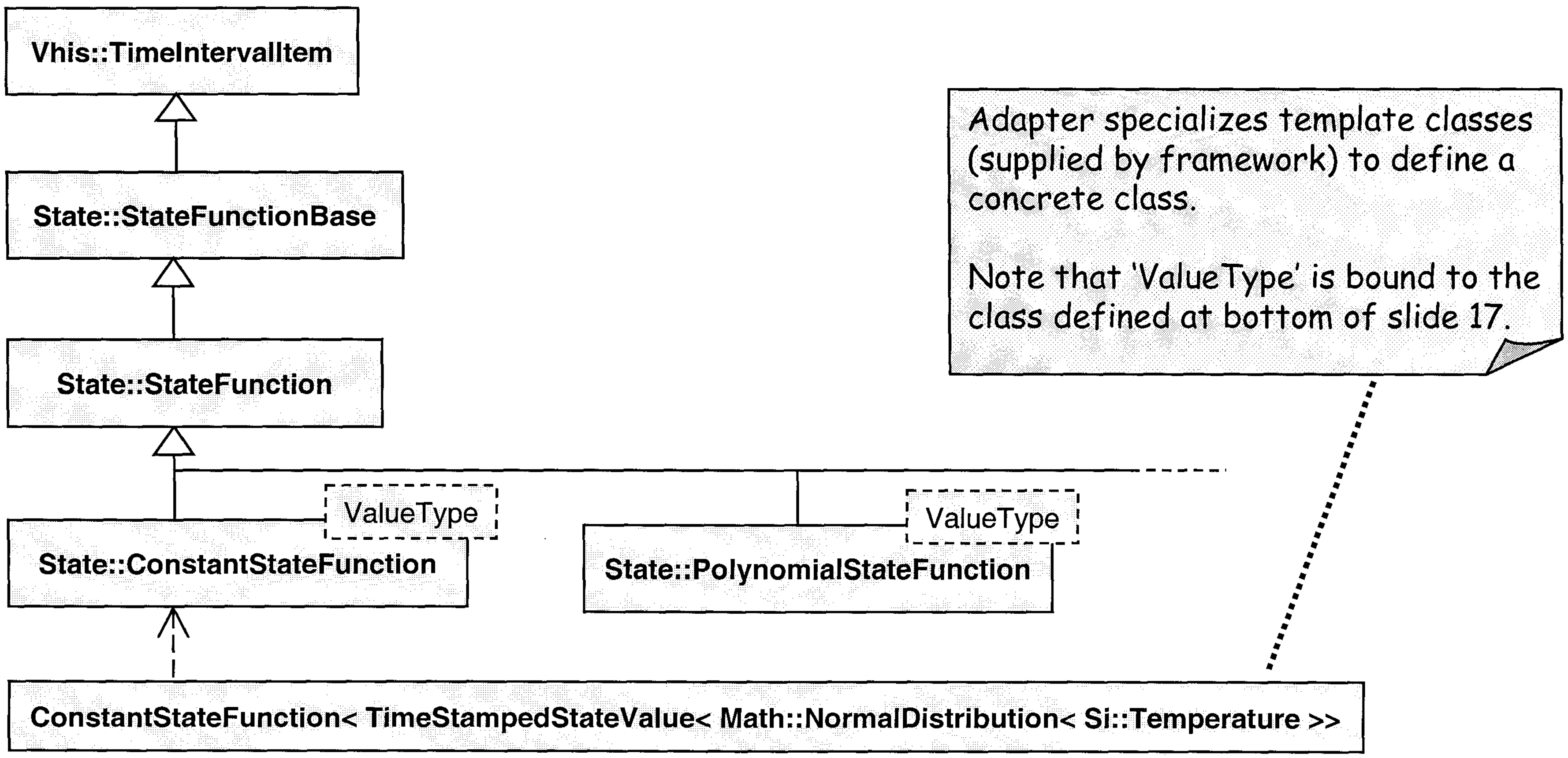
Next: Representing uncertainty

Representing Uncertainty

- An estimate must express uncertainty
- Uncertainty can be represented in many ways
  - e.g. enumerated confidence tags, variance in a Gaussian estimate, probability mass distribution over discrete states, covariance matrix, etc
  - framework does not restrict the choices
- No need to expose your internal representation and computations
  - pick what *you* want, hide the details inside *your* classes
- Only requirement is that your estimate objects be able to answer a question regarding its certainty
  - e.g. “Does the estimate have certainty $\geq c$?”
  - e.g. “Is the estimated state within range r with certainty $\geq c$?”

State Function

- *Adaptation step 2:*
  Decide how accurately to represent time-varying nature of state value
- *Example:*
  Use a constant function where dynamics are slow relative to estimation rate



State Function: Inherited Functionality

Shows attributes and operations inherited from framework classes

ConstantStateFunction< TimeStampedStateValue< Math::NormalDistribution< Si::Temperature >>

Attributes
- m_startTime : Tmgt::RTEpoch
- m_stopTime : Tmgt::RTEpoch
- m_stateValue : TimeStampedStateValue< Math::NormalDistribution< Si::Temperature > >

Normal constructor, given time interval and temperature
+ ConstantStateFunction(const RTEpoch& start, const RTEpoch& stop,
  const Math::NormalDistribution<Si::Temperature>& value)

Accessors for time interval and state value
+ getStartTime() : Tmgt::RTEpoch
+ getStopTime() : Tmgt::RTEpoch
+ getState(const RTEpoch& time) :
  RefCountP< TimeStampedStateValue< Math::NormalDistribution< Si::Temperature > >

Serialization and deserialization
+ writeObject(Ser::ObjectOutputStream&) : void
+ ConstantStateFunction(Ser::ObjectInputStream&)

Next: Time Derivatives

Representing Time Derivatives

In MDS, time derivatives of state variable *x* are represented in that same state variable

Physics:

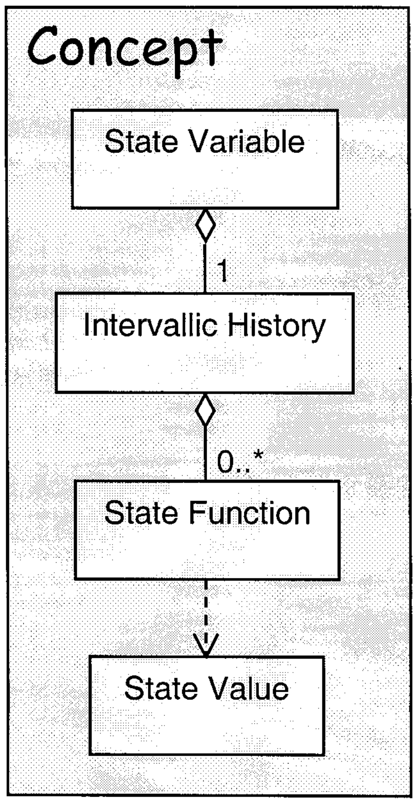
- A state refers to a physical quantity in a system
- A state value is an instantaneous description of system
- Position and velocity are separate states
- Not all time derivatives are states; acceleration not usually a state because energy is not a function of acceleration

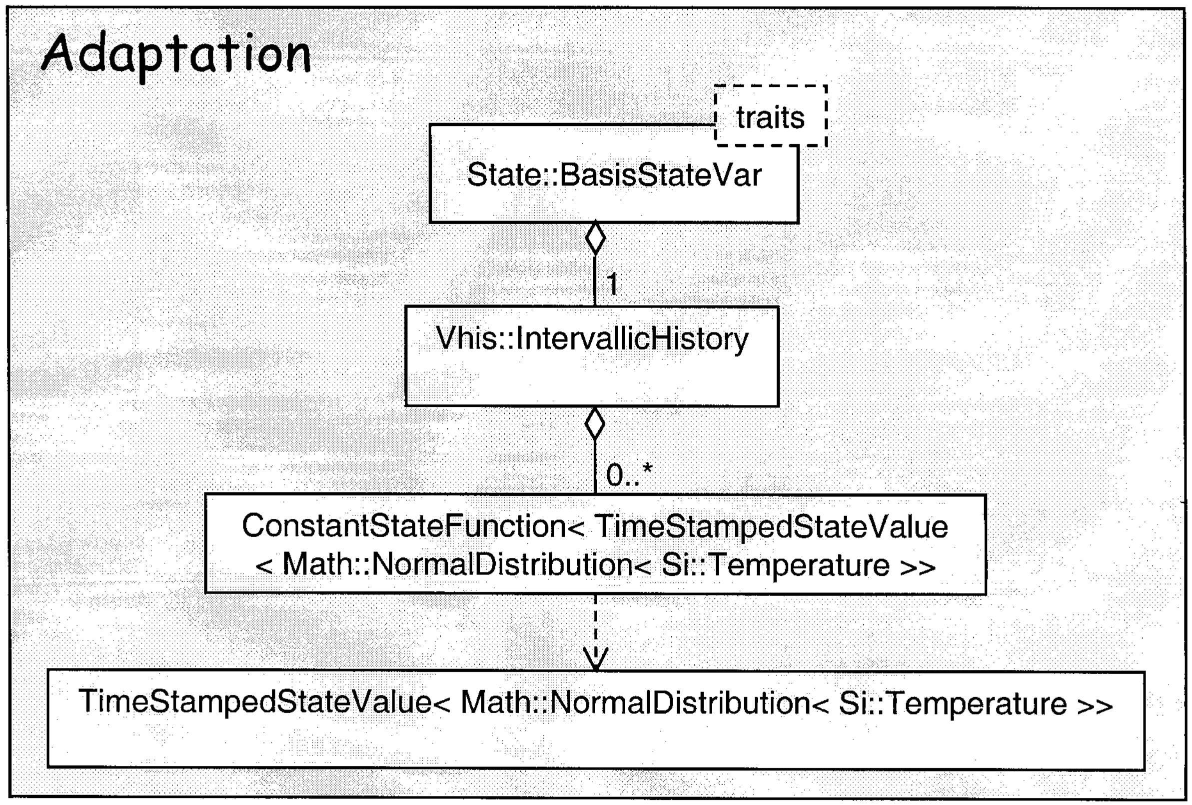
MDS:

- Because MDS maintains a history of how state changes as a function of time, derivatives are implicit, not explicit
- Can derive velocity from a history of position, so it would be redundant to have separate state variables for position and velocity
- An adapter might explicitly represent both position and velocity inside a state function, but would then have to ensure consistency between them

State Variable

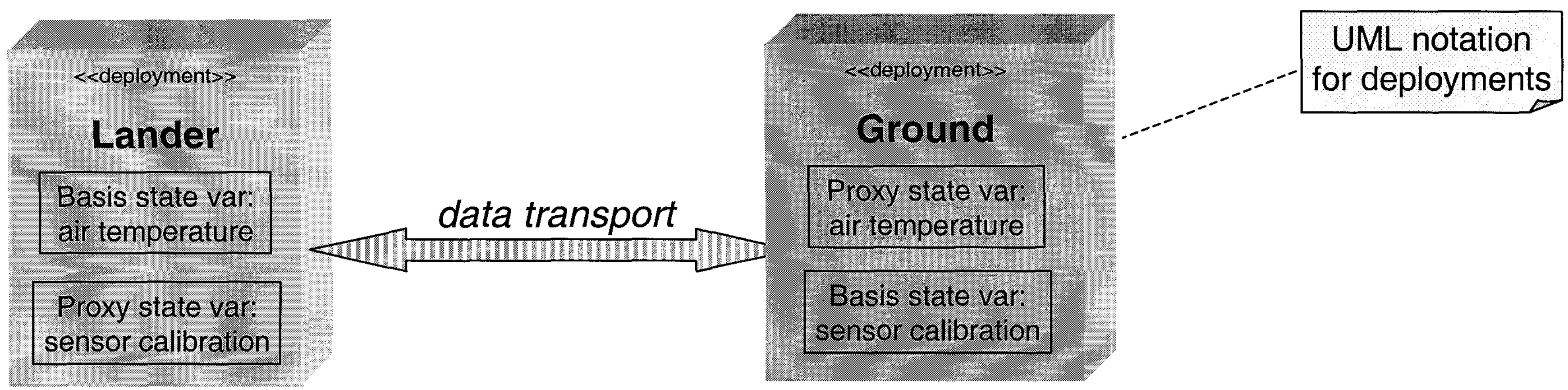
- *Adaptation step 3:*
  Define a state variable to hold state knowledge and provide access to it
- *Example:*
  Specify a camera temperature state variable that holds instances of the temperature state functions defined earlier





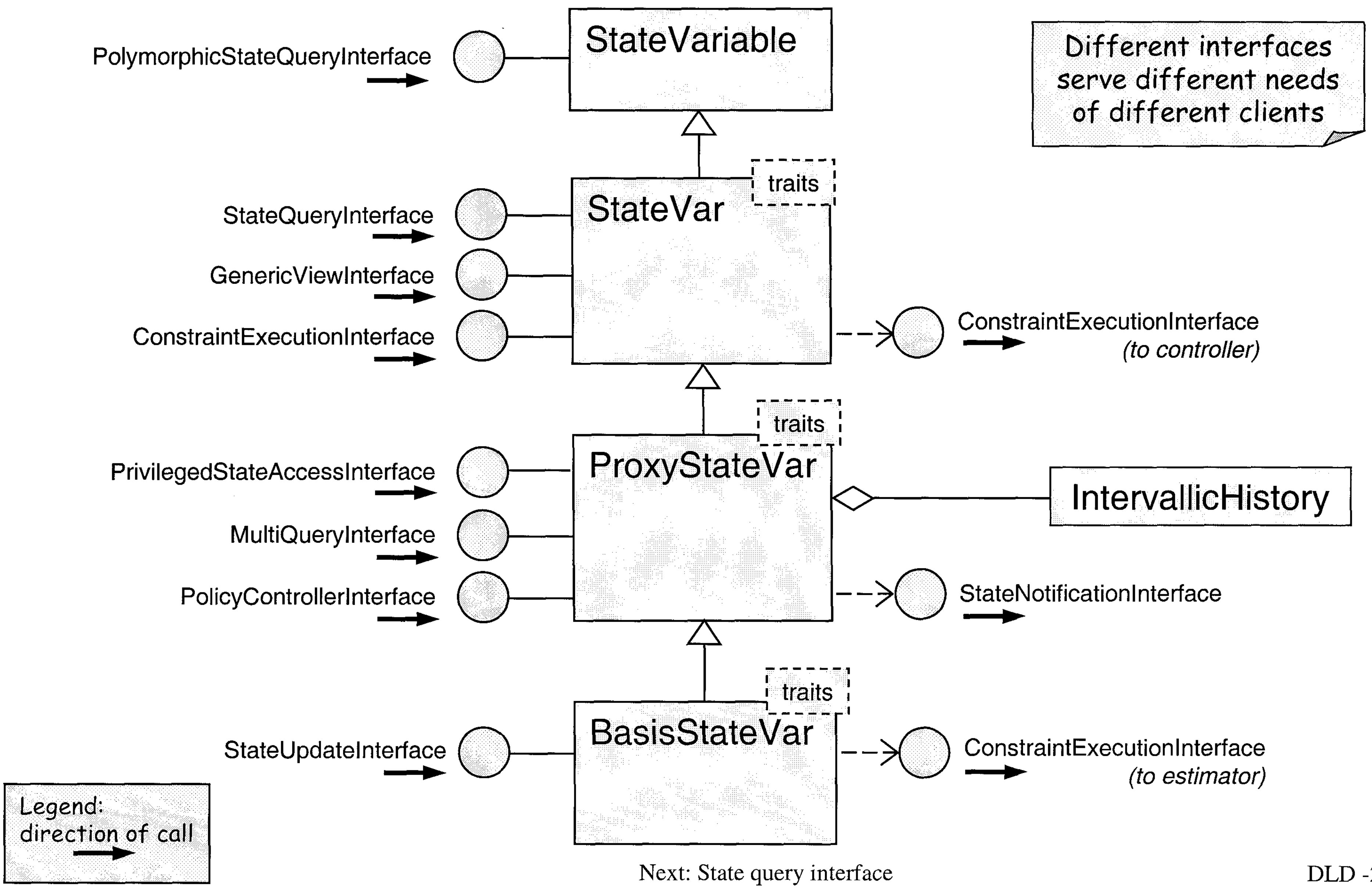
3 Kinds of State Variables

- *basis state variable* is locally estimated, near sources of evidence and ability to interpret that evidence
- *proxy state variable* provides remote, read-only, time-delayed access to value history of a basis state variable



- *derived state variable* computes a function of 2 or more state vars
  - Example: the difference between spacecraft altitude and planet surface

State Variable Interfaces



Next: State query interface

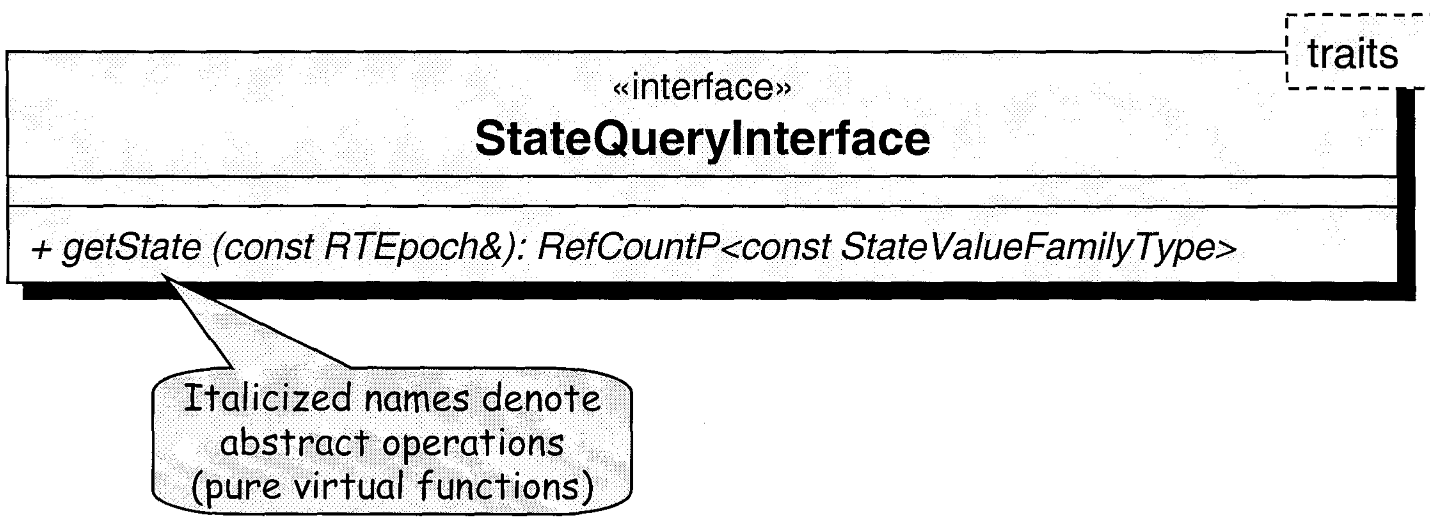
StateQueryInterface

Purpose: Provide type-safe operations for obtaining state values from any kind of state variable

Architectural rule: Queries to this interface must return "unknown" until the state variable is unlocked

Notes:

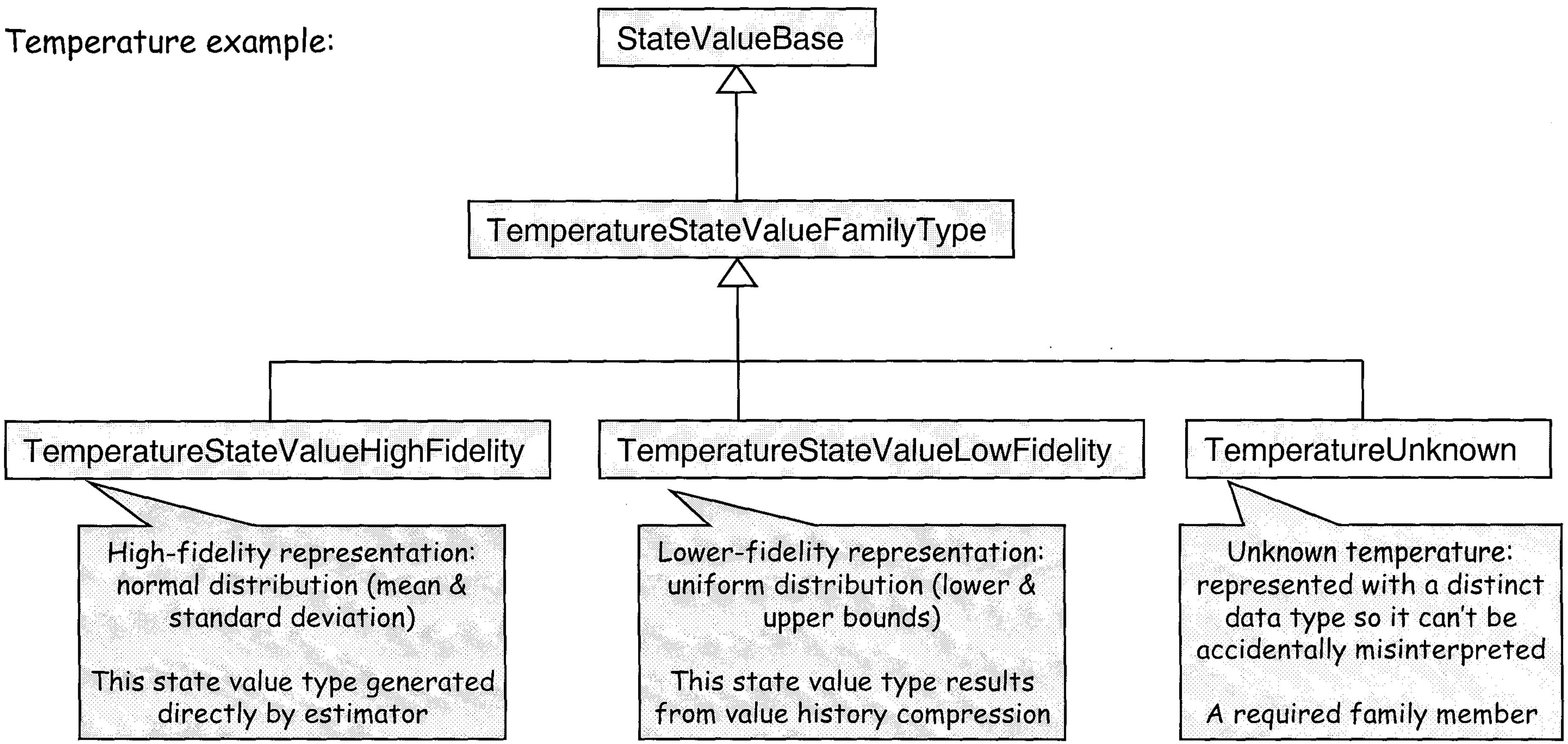
- Operation 'getState' returns a smart pointer to a heap-allocated state value object; it's general and safe
- Other operations having different speed/memory/safety tradeoffs will be added



"Family Type"

- StateQueryInterface can return more than one type of object because of value history summarization/compression
- These types are organized as members of the same family

Temperature example:



Next: state update interface

StateUpdateInterface

Purpose: Provide type-safe operations for startup initialization and routine update of value history

Architectural rule: This interface exists for exclusive use of exactly one state estimator/generator

Notes:

- value history initially locked against queries
- estimator has responsibility to:
  - selectively recover data from data catalog
  - examine/repair recovered data
  - unlock value history for queries

traits

«interface»
StateUpdateInterface

+ recoverState (const RTEpoch&, const RTEpoch&): void
+ getStateNL (const RTEpoch&): RefCountP<const StateValueFamilyType>
+ getStateFunctionNL (const RTEpoch&): RefCountP<const StateFunctionType>
+ updateState (const StateFunctionType&): void
+ unlockState (): void

StateNotificationInterface

Purpose: Enables interested listeners to be notified when a state variable's value has been updated

Design Pattern: This interface supports the 'Observer' design pattern for data-driven/event-driven reactions

Notes:

- The state variable calls this interface; it doesn't provide it
- BasisStateVar calls this during 'updateState' operation
- ProxyStateVar calls this upon receipt of new data from data transport service
- Notification includes identity of state variable and vector of changed history items

«interface»
StateNotificationInterface

+ changed (const Cmp::RefCountComponentInstance monitoredStateVar, Dm::Vhis::ValueHistory::ItemVectorRef changedItems) : void

PolicyControllerInterface

Purpose: Provides operations for setting/changing data management policies on a value history

Architectural rule: All value history-containing components must implement this interface

Notes: Data management policies control:

- when to checkpoint
- what to transport
- when to compress
- how much to recover upon startup

«interface» **PolicyControllerInterface**
+ setPolicy (const HistoryPolicy& policy) : void *+ replacePolicy (const HistoryPolicy& policy) : void* *+ revokePolicy (const Pol::PolicyActuator::PolicyIDType& policyID) : void* *+ getPolicy (const Pol::PolicyActuator::PolicyIDType& policyID) : const HistoryPolicy&*

Next: Privileged state access interface

PrivilegedStateAccessInterface

Purpose: Provide type-safe read-only access to the raw state functions contained in value history (as opposed to state values)

Notes:

- In general, state functions are *not* exposed to clients because their data representations are *implementation* choices
- Use of this interface is restricted to special cases
- Usage becomes an inspection item

traits

«interface»
PrivilegedStateAccessInterface

+ getStateFunction (const RTEpoch& time) : RefCountP<const StateFunctionType>

ConstraintExecutionInterface

Purpose: Provide operations for starting execution of a state constraint when achiever(s) ready

Notes:

- These operations are forwarded through state variable to its achievers: estimator (if present) and controller (if present)
  - Hence, state variable both *provides* and *requires* this interface
- Achiever determines readiness via combination of state constraint, state knowledge, and control model

traits

«interface»
ConstraintExecutionInterface

+ isReadyToStart (RefCountP<const StateConstraint>) : bool
+ startConstraint(RefCountP<const StateConstraint>) : void

Next: Other query interfaces

Other Query Interfaces

- **PolymorphicStateQueryInterface**:
  Provides a polymorphic query of a state variable's value, where the caller doesn't need to know the state value data type. Similar to StateQueryInterface but returns a base type for state value.

«interface»
PolymorphicQueryInterface

+ getPolyState (const RTEpoch& time) : RefCountP<StateValueBase>

- **MultiQueryInterface**:
  Provides polymorphic access to state functions to support queries of 'ground' and 'simulation' deployments by human operators. Not present in 'flight' deployments.

«interface»
MultiQueryInterface

+ getItemsInRange (const RTEpoch& start, const RTEpoch& stop) :
RefCountP<const RefCountAdapter<const std::vector<ItemRef>>>